

Simulating Reachability using First-Order Logic with Applications to Verification of Linked Data Structures

T. Lev-Ami¹, N. Immerman^{2*}, T. Reps^{3**}, M. Sagiv¹, S. Srivastava^{2*}, and
G. Yorsh^{1***}

¹ School of Comp. Sci., Tel Aviv Univ.,

{tla, msagiv, gretay}@post.tau.ac.il

² Dept. of Comp. Sci. Univ. of Massachusetts, Amherst

{immerman, siddharth}@cs.umass.edu

³ Comp. Sci. Dept., Univ. of Wisconsin, reps@cs.wisc.edu

This paper shows how to harness existing theorem provers for first-order logic to automatically verify safety properties of imperative programs that perform dynamic storage allocation and destructive updating of pointer-valued structure fields. One of the main obstacles is specifying and proving the (absence) of reachability properties among dynamically allocated cells.

The main technical contributions are methods for simulating reachability in a conservative way using first-order formulas—the formulas describe a superset of the set of program states that can actually arise. These methods are employed for semi-automatic program verification (i.e., using programmer-supplied loop invariants) on programs such as mark-and-sweep garbage collection and destructive reversal of a singly linked list. (The mark-and-sweep example has been previously reported as being beyond the capabilities of ESC/Java.)

1 Introduction

This paper explores how to harness existing theorem provers for first-order logic to prove reachability properties of programs that manipulate dynamically allocated data structures. The approach that we use involves simulating reachability in a conservative way using first-order formulas—i.e., the formulas describe a superset of the set of program states that can actually arise.

Automatically establishing safety and liveness properties of sequential and concurrent programs that permit dynamic storage allocation and low-level pointer manipulations is challenging. Dynamic allocation causes the state space to be infinite; moreover, a program is permitted to mutate a data structure by destructively updating pointer-valued fields of nodes. These features remain even if a programming language has good capabilities for data abstraction. Abstract-datatype operations are implemented using loops, procedure calls, and sequences of low-level pointer manipulations; consequently, it is hard to prove that a data-structure invariant is reestablished once a sequence of operations is finished [1]. In languages such as Java, concurrency poses yet another challenge: establishing the absence of deadlock requires establishing the absence of any cycle of threads that are waiting for locks held by other threads.

Reachability is crucial for reasoning about linked data structures. For instance, to establish that a memory configuration contains no garbage elements, we must show that

* Supported by NSF grant CCR-0207373 and a Guggenheim fellowship.

** Supported by ONR under contracts N00014-01-1-{0796,0708}.

*** Partially supported by the Israeli Academy of Science

every element is reachable from some program variable. Other cases where reachability is a useful notion include

- Specifying acyclicity of data-structure fragments, i.e., every element reachable from node n cannot reach n
- Specifying the effect of procedure calls when references are passed as arguments: only elements that are reachable from a formal parameter can be modified
- Specifying the absence of deadlocks
- Specifying safety conditions that allow establishing that a data-structure traversal terminates, e.g., there is a path from a node to a sink-node of the data structure.

The verification of such properties presents a challenge. Even simple decidable fragments of first-order logic become undecidable when reachability is added [2, 3]. Moreover, the utility of monadic second-order logic on trees is rather limited because (i) many programs allow non-tree data structures, (ii) expressing postconditions of procedures (which is essential for modular reasoning) requires referring to the pre-state that holds before the procedure executes, and thus cannot, in general, be expressed in monadic second-order logic on trees—even for procedures that manipulate only singly-linked lists, such as the in-situ list-reversal program shown in Fig. 1, and (iii) the complexity is prohibitive.

While our work was actually motivated by our experience using abstract interpretation – and, in particular, the TVLA system [4–6] – to establish properties of programs that manipulate heap-allocated data structures, in this paper, we consider the problem of verifying data-structure operations, assuming that we have user-supplied loop invariants. This is similar to the approach taken in systems like ESC/Java [7], and Pale [8].

The contributions of the paper can be summarized as follows:

Handling FO(TC) formulas using FO theorem provers. We want to use first-order theorem provers and we need to discuss the transitive closure of certain binary predicates, f . However, first-order theorem provers cannot handle transitive closure. We solve this conundrum by adding a new relation symbol f_{tc} for each such f , together with first-order axioms that assure that f_{tc} is interpreted correctly. The theoretical details of how this is done are presented in Sections 3 and 4. The fact that we are able to handle transitive closure effectively and reasonably automatically is a major contribution and quite surprising.

As explained in Section 3, the axioms that we add to control the behavior of the added predicates, f_{tc} , must be sound but not necessarily complete. One way to think about this is that we are simulating a formula, χ , in which transitive closure occurs, with a pure first-order formula χ' . If our axioms are not complete then we are allowing χ' to denote more stores than χ does. This is motivated by the fact that abstraction can be an aid in the verification of many properties; that is, a definite answer can sometimes be obtained even when information has been lost (in a conservative manner). This means that our methods are sound but potentially incomplete.

If χ' is proven valid in FO then χ is also valid in FO(TC); however, if we fail to prove that χ' is valid, it is still possible that χ is valid: the failure would be due to the incompleteness of the axioms, or the lack of time or space for the theorem prover to complete the proof.

It is easy to write a sound axiom, $T_1[f]$, that is “complete” in the very limited sense that every finite, acyclic model satisfying $T_1[f]$ must interpret f_{tc} as the reflexive, transitive closure of its interpretation of f . However, in practice this is not worth much because, as is well-known, finiteness is not expressible in first-order logic. Thus, the

properties that we want to prove do not follow from $T_1[f]$. We do prove that $T_1[f]$ is complete for positive transitive-closure properties. The real difficulties lie in proving properties involving the negation of $TC[f]$.

Induction axiom scheme. To solve the above problem, we add an induction axiom scheme. Although in general, there is no complete, recursively-enumerable axiomatization of transitive closure, we have found that on the examples we have tried, T_1 plus induction allows us to automatically prove all of our desired properties. We think of the axioms that we use as aides for the first-order theorem prover that we employ (Spass [9]) to prove the properties in question. Rather than giving Spass many instances of the induction scheme, our experience is that it finds the proof faster if we give it several axioms that are simpler to use than induction. As already mentioned, the hard part is to show that certain paths do not exist.

Coloring axiom schemes. In particular, we use three axiom schemes, having to do with partitioning memory into a small set of colors. We call instances of these schemes “coloring axioms”. Our coloring axioms are simple, and are *easily proved using Spass (in under ten seconds) from the induction axioms*. For example, the first coloring axiom scheme, **NoExit** $[A, f]$, says that if no f -edges leave color class, A , then no f -paths leave A . It turns out that the **NoExit** axiom scheme implies – and thus is equivalent to – the induction scheme. However, we have found in practice that explicitly adding other coloring axioms (which are consequences of **NoExit**) enables Spass to prove properties that it otherwise fails at.

We first assume that the programmer provides the colors by means of first-order formulas with transitive closure. Our initial experience indicates that the generated coloring axioms are useful to Spass. In particular, it provides the ability to verify programs like the mark phase of a mark-and-sweep garbage collector. This example has been previously reported as being beyond the capabilities of ESC/Java. TVLA also succeeds on this example; however our new approach provides verification methods that can in some instances be more precise than TVLA.

Prototype implementation. Perhaps most exciting, we have implemented the heuristics for selecting colors and their corresponding axioms in a prototype using Spass. We have used this to automatically choose useful color axioms and then verify several small heap-manipulating programs. More work needs to be done here, but the initial results are very encouraging.

Strengthening Nelson’s results. Greg Nelson considered a set of axiom schemes for reasoning about reachability in function graphs, i.e., graphs in which there is at most one f -edge leaving any node [10]. He left open the question of whether his axiom schemes were complete for function graphs. We show that Nelson’s axioms are provable from T_1 plus our induction axioms. We also show that Nelson’s axioms are not complete: in fact, they do not imply **NoExit**.

Outline. The remainder of the paper is organized as follows: Section 2 explains our notation and the setting; Section 3 introduces the induction axiom scheme and fills in our formal framework; Section 4 states the coloring axiom schemes; Section 5 explains the details of our heuristics; Section 6 describes some related work; Section 7 describes some future directions.

2 Preliminaries

This section defines the basic notations used in this paper and the setting.

2.1 Notation

Syntax: A relational **vocabulary** $\tau = \{p_1, p_2, \dots, p_k\}$ is a set of relation symbols, each of fixed arity. We write first-order formulas over τ with quantifiers \forall and \exists , logical connectives \wedge , \vee , \rightarrow , \leftrightarrow , and \neg , where atomic formulas include: equality, $p_i(v_1, v_2, \dots, v_{a_i})$, and $\text{TC}[f](v_1, v_2)$, where $p_i \in \tau$ is of arity a_i and $f \in \tau$ is binary. Here $\text{TC}[f](v_1, v_2)$ denotes the existence of a finite path of 0 or more f edges from v_1 to v_2 . A formula without TC is called a **first-order** formula.

We use the following precedence of logical operators: \neg has highest precedence, followed by \wedge and \vee , followed by \rightarrow and \leftrightarrow , and \forall and \exists have lowest precedence.

Semantics: A **model**, \mathcal{A} , of vocabulary τ , consists of a non-empty universe, $|\mathcal{A}|$, and a relation $p^{\mathcal{A}}$ over the universe interpreting each relation symbol $p \in \tau$. We write $\mathcal{A} \models \varphi$ to mean that the formula φ is true in the model \mathcal{A} .

2.2 Setting

We are primarily interested in formulas that arise while proving the correctness of programs. We assume that the programmer specifies pre and post-conditions for procedures and loop invariants using first-order formulas with transitive closure on binary relations. The transformer for a loop body can be produced automatically from the program code.

For instance, to establish the partial correctness with respect to a user-supplied specification of a program that contains a single loop, we need to establish three properties: First, the loop invariant must hold at the beginning of the first iteration; i.e., we must show that the loop invariant follows from the precondition and the code leading to the loop. Second, the loop invariant provided by the user must be maintained; i.e., we must show that if the loop invariant holds at the beginning of an iteration and the loop condition also holds, the transformer causes the loop invariant to hold at the end of the iteration. Finally, the postcondition must follow from the loop invariant and the condition for exiting the loop.

In general, these formulas are of the form

$$\psi_1[\tau] \wedge T[\tau, \tau'] \rightarrow \psi_2[\tau']$$

where τ is the vocabulary of the before state, τ' is the vocabulary of the after state, and T is the transformer, which may use both the before and after predicates to describe the meaning of the module to be executed. If symbol f denotes the value of a predicate before the operation then f' denotes the value of the same predicate after the operation.

An interesting special case is the proof of the maintenance formula of a loop invariant. This has the form:

$$LC[\tau] \wedge LI[\tau] \wedge T[\tau, \tau'] \rightarrow LI[\tau']$$

Here LC is the condition for entering the loop and LI is the loop invariant. $LI[\tau']$ indicates that the loop invariant remains true after the body of the loop is executed.

The challenge is that the formulas of interest contain transitive closure; thus, the validity of these formulas cannot be directly proven using a theorem prover for first-order logic.

3 Axiomatization of Transitive Closure

The original formula that we want to prove, χ , contains transitive closure, which first-order theorem provers cannot handle. To address this problem, we replace χ by the new formula, χ' , where all appearances of $\text{TC}[f]$ have been replaced by the new binary relation symbol, f_{tc} .

We show in this paper that from χ' , we can often automatically generate an appropriate first-order axiom, σ , with the following two properties:

1. if $\sigma \rightarrow \chi'$ is valid in FO then χ is valid in FO(TC).
2. A theorem prover successfully proves that $\sigma \rightarrow \chi'$ is valid in FO.

We now explain the theory behind this process. A **TC model**, \mathcal{A} , is a model such that if f and f_{tc} are in the vocabulary of \mathcal{A} , then $(f_{tc})^{\mathcal{A}} = (f^{\mathcal{A}})^*$; i.e., \mathcal{A} interprets f_{tc} as the reflexive, transitive closure of its interpretation of f .

A first-order formula φ is **TC valid** iff it is true in all TC models. We say that an axiomatization, Σ , is **TC sound** if every formula that follows from Σ is TC valid. Since first-order reasoning is sound, Σ is TC sound iff every $\sigma \in \Sigma$ is TC valid.

We say that Σ is **TC complete** if for every TC-valid φ , $\Sigma \models \varphi$. If Σ is TC complete and TC sound, then for all first-order φ ,

$$\Sigma \models \varphi \iff \varphi \text{ is TC valid}$$

Thus a TC-complete set of axioms proves exactly the first-order formulas, χ' , such that the corresponding FO(TC) formula, χ , is valid.

All the axiomatizations that we consider are TC sound. There is no recursively enumerable TC-complete axiom system (see [11, 12]).

3.1 Some TC-Sound Axioms

We begin with our first TC axiom scheme. For any binary relation symbol, f , let,

$$T_1[f] \equiv \forall u, v. f_{tc}(u, v) \leftrightarrow (u = v) \vee \exists w. f(u, w) \wedge f_{tc}(w, v)$$

We first observe that $T_1[f]$ is “complete” in a very limited way for finite, acyclic graphs, i.e., $T_1[f]$ exactly characterizes the meaning of f_{tc} for all finite, acyclic graphs. The reason this is limited, is that it does not give us a complete set of first-order axioms because, as is well known, there is no first-order axiomatization of “finite”.

Proposition 1. *Any finite and acyclic model of $T_1[f]$ is a TC model.*

Proof: Let $\mathcal{A} \models T_1[f]$ where \mathcal{A} is finite and acyclic. Let $a_0, b \in |\mathcal{A}|$. Assume there is an f -path from a_0 to b . Since $\mathcal{A} \models T_1[f]$, it is easy to see that $\mathcal{A} \models f_{tc}(a_0, b)$.

Conversely, suppose that $\mathcal{A} \models f_{tc}(a_0, b)$. If $a_0 = b$, then there is a path of length 0 from a_0 to b . Otherwise, by $T_1[f]$, there exists an $a_1 \in |\mathcal{A}|$ such that $\mathcal{A} \models f(a_0, a_1) \wedge f_{tc}(a_1, b)$. Note that $a_1 \neq a_0$ since \mathcal{A} is acyclic. If $a_1 = b$ then there is an f -path of length 1 from a to b . Otherwise there must exist an $a_2 \in |\mathcal{A}|$ such that $\mathcal{A} \models f(a_1, a_2) \wedge f_{tc}(a_2, b)$ and so on, generating a set $\{a_1, a_2, \dots\}$. None of the a_i can be equal to a_j , for $j < i$, by acyclicity. Thus, by finiteness, some $a_i = b$. Hence \mathcal{A} is a TC model. \square

Let $T'_1[f]$ be the \leftarrow direction of $T_1[f]$:

$$T'_1[f] \equiv \forall u, v. f_{tc}(u, v) \leftarrow (u = v) \vee \exists w. f(u, w) \wedge f_{tc}(w, v)$$

Proposition 2. *Let f_{tc} occur only positively in φ . If φ is TC valid, then $T'_1[f] \models \varphi$.*

Proof: Suppose that $T'_1[f] \not\models \varphi$. Let $\mathcal{A} \models T'_1[f] \wedge \neg\varphi$. Note that f_{tc} occurs only negatively in $\neg\varphi$. Furthermore, since $\mathcal{A} \models T'_1[f]$, it is easy to show by induction on the length of the path, that if there is an f -path from a to b in \mathcal{A} , then $\mathcal{A} \models f_{tc}(a, b)$. Define \mathcal{A}' to be the model formed from \mathcal{A} by interpreting f_{tc} in \mathcal{A}' as $(f^{\mathcal{A}})^*$. Thus \mathcal{A}' is a TC model and it only differs from \mathcal{A} by the fact that we have removed zero or more pairs

from $(f_{tc})^A$ to form $(f_{tc})^{A'}$. Because $\mathcal{A} \models \neg\varphi$ and f_{tc} occurs only negatively in $\neg\varphi$, it follows that $\mathcal{A}' \models \neg\varphi$, which contradicts the assumption that φ is TC valid. \square

Proposition 2 shows that proving positive facts of the form $f_{tc}(u, v)$ is easy; it is the task of proving that paths do not exist that is more subtle.

Proposition 1 shows that what we are missing, at least in the acyclic case, is that there is no first-order axiomatization of finiteness. Traditionally, when reasoning about the natural numbers, this problem is mitigated by adding induction axioms. We next introduce an induction scheme that, together with T_1 , seems to be sufficient to prove any property we need concerning TC.

Notation: In general, we will use F to denote the set of all binary relation symbols, f , such that $TC[f]$ occurs in a formula we are considering. If $\varphi[f]$ is a formula in which f occurs, let $\varphi[F] = \bigwedge_{f \in F} \varphi(f)$. Thus, for example, $T_1[F]$ is the conjunction of the axiom $T_1[f]$ for all binary relation symbols, f , under consideration.

Definition 1. For any first-order formulas $Z(u), P(u)$, and binary relation symbol, f , let the **induction principle**, $\mathbf{IND}[Z, P, f]$, be the following first-order formula:

$$(\forall z. Z(z) \rightarrow P(z)) \wedge (\forall u, v. P(u) \wedge f(u, v) \rightarrow P(v)) \\ \rightarrow \forall u, z. Z(z) \wedge f_{tc}(z, u) \rightarrow P(u)$$

The induction principle says that if every zero point satisfies P , and P is preserved when following edges, then every point reachable from a zero point satisfies P . Obviously this principle is sound.

As an easy application of the induction principle, consider the following cousin of $T_1[f]$,

$$T_2[f] \equiv \forall u, v. f_{tc}(u, v) \leftrightarrow (u = v) \vee \exists w. f_{tc}(u, w) \wedge f(w, v)$$

It is easy to see that neither of $T_1[f]$, $T_2[f]$ implies the other. However, in the presence of the induction principle they do imply each other. For example, it is easy to prove $T_2[f]$ from $T_1[f]$ using $\mathbf{IND}[Z, P, f]$ where $Z(v) \equiv v = u$ and $P(v) \equiv u = v \vee \exists w. f_{tc}(u, w) \wedge f(w, v)$. Here, for each u we use $\mathbf{IND}[Z, P, f]$ to prove by induction that every v reachable from u satisfies the right-hand side of $T_2[f]$.

A related axiom scheme that we have found useful is the transitivity of reachability:

$$\mathbf{Trans}[f] \equiv \forall u, v, w. f_{tc}(u, w) \wedge f_{tc}(w, v) \rightarrow f_{tc}(u, v)$$

4 Coloring Axioms

We next describe three TC-sound axioms schemes that are not implied by $T_1[F] \wedge T_2[F]$, and are provable from the induction principle of Section 3. We will see in the sequel that these coloring axioms are very useful in proving that paths do not exist, permitting us to verify a variety of algorithms. In Section 5, we will present some heuristics for automatically choosing particular instances of the coloring axiom schemes that enable us to prove our goal formulas.

The first coloring axiom scheme is the NoExit axiom scheme:

$$(\forall u, v. A(u) \wedge \neg A(v) \rightarrow \neg f(u, v)) \rightarrow \forall u, v. A(u) \wedge \neg A(v) \rightarrow \neg f_{tc}(u, v) \quad (1)$$

for any first-order formula $A(u)$, and binary relation symbol, f , **NoExit** $[A, f]$ says that if no f -edge leaves color class A , then no f -path leaves color class A .

Observe that although it is very simple, **NoExit** $[A, f]$ does not follow from $T_1[f] \wedge T_2[f]$. Let $G_1 = (V, f, f_{tc}, A)$ consist of two disjoint cycles: $V = \{1, 2, 3, 4\}$, $f = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 3, 4 \rangle, \langle 4, 3 \rangle\}$, and $A = \{1, 2\}$. Let f_{tc} have all 16 possible edges. Thus G_1 satisfies $T_1[f] \wedge T_2[f]$ but violates **NoExit** $[A, f]$. Even for acyclic models, **NoExit** $[A, f]$ does not follow from $T_1[f] \wedge T_2[f]$ because there are infinite models in which the implication does not hold (see [12]).

NoExit $[A, f]$ follows easily from the induction principle: if no edges leave A , then induction tells us that everything reachable from a point in A satisfies A . Similarly, **NoExit** $[A, f]$ implies the induction axiom, **IND** $[Z, A, f]$, for any formula Z .

The second coloring axiom scheme is the **GoOut** axiom: for any first-order formulas $A(u)$, $B(u)$, and binary relation symbol, f , **GoOut** $[A, B, f]$ says that if the only edges leaving color class A are to B , then any path from a point in A to a point not in A must pass through B .

$$\begin{aligned} (\forall u, v. A(u) \wedge \neg A(v) \wedge f(u, v) \rightarrow B(v)) \rightarrow \\ \forall u, v. A(u) \wedge \neg A(v) \wedge f_{tc}(u, v) \rightarrow \exists b. B(b) \wedge f_{tc}(u, b) \wedge f_{tc}(b, v) \end{aligned} \quad (2)$$

To see that **GoOut** $[A, B, f]$ follows from the induction principle, assume that the only edges out of A enter B . For any fixed u in A , we prove by induction that any point v reachable from u is either in A or has a predecessor, b in B , that is reachable from u .

The third coloring axiom scheme is the **NewStart** axiom, which is useful in the context of dynamically changing graphs: for any first-order formula $A(u)$, and binary relation symbols f and g , think of f as the previous edge relation and g as the current edge relation. **NewStart** $[A, f, g]$ says that if there are no new edges between A nodes, then any new path from A must leave A to make its change:

$$\begin{aligned} (\forall u, v. A(u) \wedge A(v) \wedge g(u, v) \rightarrow f(u, v)) \rightarrow \\ \forall u, v. g_{tc}(u, v) \wedge \neg f_{tc}(u, v) \rightarrow \exists b. \neg A(b) \wedge g_{tc}(u, b) \wedge g_{tc}(b, v) \end{aligned} \quad (3)$$

NewStart $[A, f, g]$ follows from the induction principle by a proof that is similar to the proof of **GoOut** $[A, B, f]$

We remark that the spirit behind our consideration of the coloring axioms is similar to that found in a paper of Greg Nelson's in which he introduced a set of reachability axioms for a functional predicate, f , i.e., there is at most one f edge leaving any point [10]. Nelson asked whether his axiom schemes are complete for the functional setting. We remark that Nelson's axiom schemes are provable from T_1 plus our induction principle. However, Nelson's axiom schemes are not complete: we constructed a functional graph satisfying Nelson's axioms but violating **NoExit** $[A, f]$, (see [12]).

At least one of Nelson's axiom schemes does seem orthogonal to our coloring axioms and may be useful in certain proofs. Nelson's fifth axiom scheme states that the points reachable from a given point are linearly ordered. The soundness of the axiom scheme is due to the fact that f is functional. We make use of a simplified version of Nelson's ordering axiom scheme: Let **Func** $[f] \equiv \forall u, v, w. f(u, v) \wedge f(u, w) \rightarrow v = w$; then,

$$\mathbf{Order}[f] \equiv \mathbf{Func}[f] \rightarrow \forall u, v, w. f_{tc}(u, v) \wedge f_{tc}(u, w) \rightarrow f_{tc}(v, w) \vee f_{tc}(w, v)$$

5 Heuristics for Using the Coloring Axioms

This section presents heuristics for using the coloring axioms. Toward that end, it answers the following questions:

- How can the coloring axioms be used by a theorem prover to prove χ ?
- When should a specific instance of a coloring axiom be given to the theorem prover while trying to prove χ ?
- What part of the process can be automated?

We first present a running example that will be used in later sections to illustrate the heuristics. We then explain how the coloring axioms are useful, describe the search space for useful axioms, give an algorithm for exploring this space, and conclude by discussing a prototype implementation we have developed that proves the example presented and others.

5.1 Reverse Specification

The heuristics described in Sections 5.2–5.4 are illustrated on problems that arise in the verification of partial correctness of a list reversal procedure. Other examples proven using this technique can be found in the full version of this paper [12].

The procedure `reverse`, shown in Fig. 1, performs in-place reversal of a singly linked list, destructively updating the list. The precondition requires that the input list be acyclic and unshared. For simplicity, we assume that there is no garbage. The postcondition ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter on entry to `reverse` are exactly the nodes reachable from the return value of `reverse` at the exit. Most importantly, it ensures that each edge in the original list is reversed in the returned list.

The specification for `reverse` is shown in Fig. 2. We use unary predicates to represent program variables and binary predicates to represent data-structure fields. Fig. 2(a) defines some shorthands. To specify that a unary predicate z can point to a single node at a time and that a binary predicate f of a node can point to at most one node (a partial function), we use *unique*[z] and *func*[f]. To specify that there are no cycles of f -fields in the graph, we use *acyclic*[f]. To specify that the graph does not contain nodes shared by f -fields, (i.e., nodes with 2 or more incoming f -fields), we use *unshared*[f]. To specify that all nodes in the graph are reachable from z_1 or z_2 by following f -fields, we use *total*[z_1, z_2, f]. Another helpful shorthand is $r_{x,f}(v)$ which specifies that v is reachable from the node pointed to by x using f -edges.

The precondition of the `reverse` procedure is shown in Fig. 2(b). We use the predicates *xe* and *ne* to record the values of the variable x and the next field at the beginning of the procedure. The precondition requires that the list pointed to by x be acyclic and unshared. It also requires that *unique*[z] and *func*[f] hold for all unary predicates z that represent program variables and all binary predicates f that represent fields, respectively. For simplicity, we assume that there is no garbage, i.e., all nodes are reachable from x .

```
Node reverse(Node x){
  [0] Node y = null;
  [1] while (x != null){
  [2]   Node t = x.next;
  [3]   x.next = y;
  [4]   y = x;
  [5]   x = t;
  [6] }
  [7] return y;
}
```

Fig. 1. A simple Java-like implementation of the in-place reversal of a singly-linked list.

The post-condition is shown in Fig. 2(c). It ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter x on entry to the procedure are exactly the nodes reachable from the return value y at the exit. Most importantly, we wish to show that each edge in the original list is reversed in the returned list (see Eq. (11)).

A loop invariant is given in Fig. 2(d). It describes the state of the program at the beginning of each loop iteration. Every node is in one of two disjoint lists pointed by x and y (Eq. (12)). The lists are acyclic and unshared. Every edge in the list pointed to by x is exactly an edge in the original list (Eq. (14)). Every edge in the list pointed to by y is the reverse of an edge in the original list (Eq. (15)). The only original edge going out of y is to x (Eq. (16)).

The transformer is given in Fig. 2(e), using the primed predicates n' , x' , and y' to describe the values of predicates n , x , and y , respectively, at the end of the iteration.

(a)	$unique[z] \stackrel{\text{def}}{=} \forall v_1, v_2. z(v_1) \wedge z(v_2) \rightarrow v_1 = v_2$	(4)
	$func[f] \stackrel{\text{def}}{=} \forall v_1, v_2, v. f(v, v_1) \wedge f(v, v_2) \rightarrow v_1 = v_2$	(5)
	$acyclic[f] \stackrel{\text{def}}{=} \forall v_1, v_2. \neg f(v_1, v_2) \vee \neg TC[f](v_2, v_1)$	(6)
	$unshared[f] \stackrel{\text{def}}{=} \forall v_1, v_2, v. f(v_1, v) \wedge f(v_2, v) \rightarrow v_1 = v_2$	(7)
	$total[z_1, z_2, f] \stackrel{\text{def}}{=} \forall v. \exists w. (z_1(w) \vee z_2(w)) \wedge TC[f](w, v)$	(8)
	$r_{x,f}(v) \stackrel{\text{def}}{=} \exists w. x(w) \wedge TC[f](w, v)$	(9)
(b)	$pre \stackrel{\text{def}}{=} total[xe, xe, ne] \wedge acyclic[ne] \wedge unshared[ne] \wedge unique[xe] \wedge func[ne]$	(10)
(c)	$post \stackrel{\text{def}}{=} total[y, y, n] \wedge acyclic[n] \wedge unshared[n] \wedge \forall v_1, v_2. ne(v_1, v_2) \leftrightarrow n(v_2, v_1)$	(11)
(d)	$LI[x, y, n] \stackrel{\text{def}}{=} total[x, y, n] \wedge \forall v. (\neg r_{x,n}(v) \vee \neg r_{y,n}(v)) \wedge$	(12)
	$acyclic[n] \wedge unshared[n] \wedge unique[x] \wedge unique[y] \wedge func[n] \wedge$	(13)
	$\forall v_1, v_2. (r_{x,n}(v_1) \rightarrow (ne(v_1, v_2) \leftrightarrow n(v_1, v_2))) \wedge$	(14)
	$\forall v_1, v_2. (r_{y,n}(v_1) \wedge \neg y(v_1) \rightarrow (ne(v_1, v_2) \leftrightarrow n(v_2, v_1))) \wedge$	(15)
	$\forall v_1, v_2, v. y(v_1) \rightarrow (x(v_2) \leftrightarrow ne(v_1, v_2))$	(16)
(e)	$T \stackrel{\text{def}}{=} \forall v. (y'(v) \leftrightarrow x(v)) \wedge \forall v. (x'(v) \leftrightarrow \exists w. x(w) \wedge n(w, v)) \wedge$	
	$\forall v_1, v_2. n'(v_1, v_2) \leftrightarrow ((n(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge y(v_2)))$	(17)

Fig. 2. Example specification of reverse procedure: (a) shorthands, (b) precondition pre , (c) postcondition $post$, (d) loop invariant $LI[x, y, n]$, (e) transformer T (effect of the loop body).

5.2 Proving Formulas using the Coloring Axioms

All the coloring axioms have the form $A \equiv P_A \rightarrow C_A$, where P_A and C_A are closed formulas. We call P_A the axiom's premise and C_A the axiom's conclusion. For an axiom to be useful, the theorem prover will have to prove the premise (as a subgoal) and then use the conclusion in the proof of the goal formula χ . For each of the coloring axioms, we now explain when the premise can be proved, how its conclusion can help, and give an example.

NoExit. The premise $P_{\text{NoExit}}[C, f]$ states that there are no f -edges exiting color class C . When C is a unary predicate appearing in the program, the premise is some-

times a direct result of the loop invariant. Another color that will be used heavily throughout this section is reachability from a unary predicate, i.e., unary reachability, formally defined in Eq. (9). Let's examine two cases. $P_{\text{NoExit}}[r_{x,f}, f]$ is immediate from the definition of $r_{x,f}$ and the transitivity of f_{tc} . $P_{\text{NoExit}}[r_{x,f}, f']$ actually states that there is no f path from x to an edge for which f' holds but f doesn't, i.e., a change in f' with respect to f . Thus, we use the absence of f -paths to prove the absence of f' -paths. In many cases, the change is an important part of the loop invariant, and paths from and to it are part of the specification.

A sketch of the proof by refutation of $P_{\text{NoExit}}[r_{x',n}, n']$ that arises in the reverse example is given in Fig. 3. The numbers in brackets are the stages of the proof.

1. The negation of the premise expands to:

$$\exists u_1, u_2, u_3. x'(u_1) \wedge n_{tc}(u_1, u_2) \wedge \neg n_{tc}(u_1, u_3) \wedge n'(u_2, u_3)$$

2. Since u_2 is reachable from u_1 and u_3 is not, by transitivity of n_{tc} , we have $\neg n(u_2, u_3)$.
3. By the definition of n' in the transformer, the only edge in which n differs from n' is out of x (one of the clauses generated from Eq. (17) is $\forall v_1, v_2. \neg n'(v_1, v_2) \vee \neg n(v_1, v_2) \vee x(v_1)$). Thus, $x(u_2)$ holds.
4. By the definition of x' it has an incoming n edge from x . Thus, $n(u_2, u_1)$ holds. The list pointed to by x must be acyclic, whereas we have a cycle between u_1 and u_2 ; i.e., we have a contradiction. Thus, $P_{\text{NoExit}}[r_{x',n}, n']$ must hold.

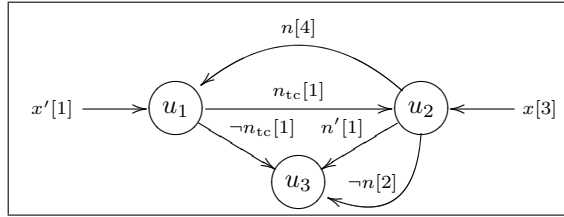


Fig. 3. Proving $P_{\text{NoExit}}[r_{x,n}, n']$.

$C_{\text{NoExit}}[C, f]$ states there are no f paths (f_{tc} edges) exiting C . This is useful because proving the absence of paths is the difficult part of proving formulas with TC.

GoOut. The premise $P_{\text{GoOut}}[A, B, f]$ states that all f edges going out of color class A , go to B . When A and B are unary predicates that appear in the program, again the premise sometimes holds as a direct result of the loop invariant. An interesting special case is when B is defined as $\exists w. A(w) \wedge f(w, v)$. In this case the premise is immediate. Note that in this case the conclusion is provable also from T_1 . However, from experience, the axiom is very useful for improving performance (2 orders of magnitude when proving the acyclic part of reverse's post condition).

$C_{\text{GoOut}}[A, B, f]$ states that all paths out of A must pass through B . Thus, under the premise $P_{\text{GoOut}}[A, B, f]$, if we know that there is a path from A to somewhere outside of A , we know that there is a path to there from B . In case all nodes in B are reachable from all nodes in A , together with the transitivity of f_{tc} this means that the nodes reachable from B are exactly the nodes outside of A that are reachable from A .

For example, $C_{\text{GoOut}}[y', y, n']$ allows us to prove that only the original list pointed to by y is reachable from y' (in addition to y' itself).

NewStart. The premise $P_{\text{NewStart}}[C, g, h]$ states that all g edges between nodes in C are also h edges. This can mean the iteration has not added edges or has not removed

edges according to the selection of h and g . In some cases, the premise holds as a direct result of the definition of C and the loop invariant.

$C_{\text{NewStart}}[C, g, h]$ means that every g path that is not an h path must pass outside of C . Together with $C_{\text{NoExit}}[C, g]$, it proves there are no new paths within C .

For example, in reverse the **NewStart** scheme can be used as follows. No outgoing edges were added to nodes reachable from y . There are no n or n' edges from nodes reachable from y to nodes not reachable from y . Thus, no paths were added between nodes reachable from y . Since the list pointed to by y is acyclic before the loop body, we can prove that it is acyclic at the end of the loop body.

We can see that **NewStart** allows the theorem prover to reason about paths within a color, and the other axioms allow the theorem prover to reason about paths between colors. Together, given enough colors, the theorem prover can often prove all the facts that it needs about paths and thus prove the formula of interest.

5.3 The Search Space of Possible Axioms

To answer the question of when we should use a specific instance of a coloring axiom when attempting to prove the target formula, we first define the search space in which we are looking for such instances. The axioms can be instantiated with the colors defined by an arbitrary unary formula (one free variable) and one or two binary predicates. First, we limit ourselves to binary predicates for which TC was used in the target formula. Now, since it is infeasible to consider all arbitrary unary formulas, we start limiting the set of colors we consider.

The initial set of colors to consider are unary predicates that occur in the formula we want to prove. Interestingly enough, these colors are enough to prove that the post-condition of mark and sweep is implied by the loop invariant, because the only axiom we need is **NoExit**[marked, f].

An immediate extension that is very effective is reachability from unary predicates, as defined in Eq. (9). Instantiating all possible axioms from the unary predicates appearing in the formula and their unary reachability predicates, allows us to prove reverse. For a list of the axioms needed to prove reverse, see Fig. 4. Other example are presented in [12]. Finally, we consider Boolean combinations of the above colors. Though not used in the examples shown in this paper, this is needed, for example, in the presence of sharing or when splicing two lists together.

NoExit [$r_{x',n}, n'$]	GoOut [x, x', n]	NewStart [$r_{x',n}, n, n'$]	NewStart [$r_{x',n}, n', n$]
NoExit [$r_{x',n'}, n$]	GoOut [x, y, n']	NewStart [$r_{x',n'}, n, n'$]	NewStart [$r_{x',n'}, n', n$]
NoExit [$r_{y,n}, n'$]		NewStart [$r_{y,n}, n, n'$]	NewStart [$r_{y,n}, n', n$]
NoExit [$r_{y,n'}, n$]		NewStart [$r_{y,n'}, n, n'$]	NewStart [$r_{y,n'}, n', n$]

Fig. 4. The instances of coloring axioms used in proving reverse.

All the colors above are based on the unary predicates that appear in the original formula. To prove the reverse example, we needed x' as part of the initial colors. Table 1 gives a heuristic for finding the initial colors we need in cases when they cannot be deduced from the formula, and how it applies to reverse

An interesting observation is that the initial colors we need can, in many cases, be deduced from the program code. As in the previous section, we have a good way for deducing paths between colors and within colors in which the edges have not changed. The program usually manipulates fields using pointers, and can traverse an edge only in

one direction. Thus, the unary predicates that represent the program variables (including the temporary variables) are in many cases what we need as initial colors.

Group	Criteria
Roots[f]	All changes are reachable from one of the colors using f_{tc}
StartChange[f,g]	All edges for which f and g differ start from a node in these colors
EndChange[f,g]	All edges for which f and g differ end at a node in these colors

(a)

Group	Colors	Group	Colors
Roots[n]	$x(v), y(v)$	StartChange[n, n']	$x(v)$
Roots[n']	$x'(v), y'(v)$	EndChange[n, n']	$y(v), x'(v)$

(b)

Table 1. (a) Heuristic for choosing initial colors. (b) Results of applying the heuristic on reverse.

5.4 Exploring the Search Space

When trying to automate the process of choosing colors, the problem is that the set of possible colors to choose from is doubly-exponential in the number of initial colors; giving all the axioms directly to the theorem prover is infeasible. In this section, we define a heuristic algorithm for exploring a limited number of axioms in a directed way. Pseudocode for this algorithm is shown in Fig. 5. The operator \vdash is implemented as a call to a theorem prover.

<pre> explore(Init, χ) { Let $\chi = \psi \rightarrow \varphi$ $\Sigma := \{\text{Trans}[f], \text{Order}[f] \mid f \in F\}$ $\Sigma := \Sigma \cup \{T_1[f], T_2[f] \mid f \in F\}$ $C := \{r_{c,f}(v) \mid c \in \text{Init}, f \in F\}$ $C := C \cup \text{Init}$ $i := 1$ forever { $C' := BC(i, C)$ phase1(C', Σ, ψ) phase2(C', Σ, ψ) phase3(Σ, ψ) if $\Sigma \wedge \psi \vdash \varphi$ return SUCCESS $i := i + 1$ } } </pre>	<pre> phase1(C, Σ, ψ) { foreach $f \in F, c_s \neq c_e \in C$ if $\Sigma \wedge \psi \vdash P_{\text{GoOut}}[c_s, c_e, f]$ $\Sigma := \Sigma \cup \{C_{\text{GoOut}}[c_s, c_e, f]\}$ } phase2(C, Σ, ψ) { foreach $f \in F, c \in C$ if $\Sigma \wedge \psi \vdash P_{\text{NoExit}}[c, f]$ $\Sigma := \Sigma \cup \{C_{\text{NoExit}}[c, f]\}$ } phase3(Σ, ψ) { foreach $C_{\text{NoExit}}[c, f] \in \Sigma, g \neq f \in F$ if $\Sigma \wedge \psi \vdash P_{\text{NewStart}}[c, f, g]$ $\Sigma := \Sigma \cup \{C_{\text{NewStart}}[c, f, g]\}$ } </pre>
---	--

Fig. 5. An iterative algorithm for instantiating the axiom schemes. Each iteration consists of three phases that augment the axiom set Σ

Because the coloring axioms have the form $A \equiv P_A \rightarrow C_A$, the theorem prover must prove P_X or the axiom is of no use. Therefore, the pseudocode works iteratively, trying to prove P_A from the current $\psi \wedge \Sigma$, and if successful it adds C_A to Σ .

The algorithm tries colors in increasing levels of complexity. $BC(i, C)$ gives all the Boolean combinations of the predicates in C up to size i . After each iteration we try to

prove the goal formula. Sometimes we need the conclusion of one axiom to prove the premise of another. The **NoExit** axioms are particularly useful for proving P_{NewStart} . Therefore, we need a way to order instantiations so that axioms useful for proving the premises of other axioms are acquired first. The ordering we chose is based on phases: First, try to instantiate axioms from the axiom scheme **GoOut**. Second, try to instantiate axioms from the axiom scheme **NoExit**. Finally, try to instantiate axioms from the axiom scheme **NewStart**. For $\text{NewStart}[c, f, g]$ to be useful, we need to be able to show that there are either no incoming f paths or no outgoing f paths from c . Thus, we only try to instantiate such an axiom when either $P_{\text{NoExit}}[c, f]$ or $P_{\text{NoExit}}[\neg c, f]$ were proven.

5.5 Implementation

The algorithm presented here was implemented using a Perl script and the Spass theorem prover [9] and used successfully to verify the example programs of Section 5.1.

The method described above can be optimized. For instance, if C_A has already been added to the axioms, we do not try to prove P_A again. These details are important in practice, but have been omitted for brevity.

When trying to prove the different premises, Spass may fail to terminate if the formula that it is trying to prove is invalid. Thus, we limit the time that Spass can spend proving each formula. It is possible that we will fail to acquire useful axioms this way.

6 Related Work

Shape Analysis. This work was motivated by our experience with TVLA [4, 5], which is a generic system for abstract interpretation [13]. The TVLA system is more automatic than the methods described in this paper since it does not rely on user-supplied loop invariants. However, the techniques presented in the present paper are potentially more precise due to the use of full first-order reasoning. It can be shown that the **NoExit** scheme allows to infer reachability at least as precisely as evaluation rules for 3-valued logic with Kleene semantics. In the future, we hope to develop an efficient non-interactive theorem prover that enjoys the benefits of both approaches. An interesting observation is that the colors needed in our examples to prove the formula are the same unary predicates used by TVLA to define its abstraction. This similarity may, in the future, help us find better ways to automatically instantiate the required axioms. In particular, inductive logic programming has recently been used to learn formulas to use in TVLA abstractions [14], which holds out the possibility of applying similar methods to further automate the approach of the present paper.

Decidable Logics. Decidable logics can be employed to define properties of linked data structures: Weak monadic second-order logic has been used in [15, 8] to define properties of heap-allocated data structures, and to conduct Hoare-style verification using programmer-supplied loop invariants in the PALE system [8]. A decidable logic called L_r (for “logic of reachability expressions”) was defined in [16]. L_r is rich enough to express the shape descriptors studied in [17] and the path matrices introduced in [18].

The present paper does not develop decision procedures, but instead suggests methods that can be used in conjunction with existing theorem provers. Thus, the techniques are incomplete and the theorem provers need not terminate. However, our initial experience is that the extra flexibility gained by the use of first-order logic with transitive closure is promising. For example, we can prove the correctness of imperative destructive

list-reversal specified in a natural way and the correctness of mark and sweep garbage collectors, which are beyond the scope of Mona and L_r .

Indeed, in [19], we have tried to simulate existing data structures using decidable logics and realized that this can be tricky because the programmer may need to prove a specific simulation invariant for a given program. Giving an inaccurate simulation invariant causes the simulation to be unsound. One of the advantages of the technique described in the present paper is that soundness is guaranteed no matter which axioms are instantiated. Moreover, the simulation requirements are not necessarily expressible in the decidable logic.

Other First-Order Axiomatizations of Linked Data Structures. The closest approach to ours that we are aware of was taken by Nelson as we describe in the full version of the paper [12]. This also has some follow-up work by Leino and Joshi [20]. Our impression from their write-up is that Leino and Joshi’s work can be pushed forward by using our coloring axioms.

Dynamic Maintenance of Transitive Closure. Another orthogonal but promising approach to transitive closure is to maintain reachability relations incrementally as we make unit changes in the data structure. It is known that in many cases, reachability can be maintained by first-order formulas [21, 22] and even sometimes by quantifier-free formulas [23]. Furthermore, in these cases, it is often possible to automatically derive the first-order update predicates using finite differencing [24].

7 Conclusion

This paper reports on our initial attempts at applying the methodology that has been described; hence, only preliminary conclusions can be drawn.

As mentioned earlier, proving the absence of paths is the difficult part of proving formulas with TC. The promise of the approach is that it is able to handle such formulas effectively and reasonably automatically, as shown by the fact that it can successfully handle the programs described in Section 5 and the full version of the paper [12]. Many issues remain for further work, such as,

- Establishing whether $T_1[F]$ plus the induction scheme is complete for interesting subclasses of formulas (e.g. functional graphs).
- Exploring other heuristics for identifying color classes.
- Exploring variations of the algorithm given in Fig. 5 for instantiating coloring axioms.
- Exploring the use of additional axiom schemes, such as two of the schemes from [10], which are likely to be useful when dealing with predicates that are partial functions. Such predicates arise in programs that manipulate singly-linked or doubly-linked lists—or, more generally, data structures that are acyclic in one or more “dimensions” [25] (i.e., in which the iterated application of a given field selector can never return to a previously visited node).

Thanks to Aharon Abadi and Roman Manevich for interesting suggestions.

References

1. Hoare, C.: Recursive data structures. *Int. J. of Comp. and Inf. Sci.* **4** (1975) 105–132
2. Grädel, E., M. Otto, E. Rosen: Undecidability results on two-variable logics. *Archive of Math. Logic* **38** (1999) 313–354
3. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability of transitive closure logics. In: *CSL’04*. (2004)

4. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Static Analysis Symp. (2000) 280–301
5. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst. (2002)
6. Reps, T., Sagiv, M., Wilhelm, R.: Static program analysis via 3-valued logic. In: CAV. (2004) 15–30
7. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for java. In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2002)
8. Møller, A., Schwartzbach, M.: The pointer assertion logic engine. In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2001) 221–231
9. Weidenbach, C., Gaede, B., Rock, G.: Spass & flotter version 0.42. In: CADE-13: Proceedings of the 13th International Conference on Automated Deduction, Springer-Verlag (1996) 141–145
10. Nelson, G.: Verifying reachability invariants of linked structures. In: Symp. on Princ. of Prog. Lang. (1983) 38–47
11. Avron, A.: Transitive closure and the mechanization of mathematics. In: Thirty Five Years of Automating Mathematics, Kluwer Academic Publishers (2003) 149–171
12. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. Available at “<http://www.cs.tau.ac.il/~tla/2005/papers/cade05full.pdf>” (2005)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL ’77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1977) 238–252
14. Loginov, A., Reps, T., Sagiv, M.: Abstraction refinement via inductive learning. In: Proc. Computer-Aided Verif. (2005)
15. Elgaard, J., Møller, A., Schwartzbach, M.: Compile-time debugging of C programs working on trees. In: European Symp. On Programming. (2000) 119–134
16. Benedikt, M., Reps, T., Sagiv, M.: A decidable logic for describing linked data structures. In: European Symp. On Programming. (1999) 2–19
17. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. Trans. on Prog. Lang. and Syst. **20** (1998) 1–50
18. Hendren, L.: Parallelizing Programs with Recursive Data Structures. PhD thesis, Cornell Univ., Ithaca, NY (1990)
19. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: Verification via structure simulation. In: Proc. Computer-Aided Verif. (2004) 281–294
20. Leino, R.: Recursive object types in a logic of object-oriented programs. Nordic J. of Computing **5** (1998) 330–360
21. Dong, G., Su, J.: Incremental and decremental evaluation of transitive closure by first-order queries. Inf. & Comput. **120** (1995) 101–106
22. Patnaik, S., Immerman, N.: Dyn-FO: A parallel, dynamic complexity class. Journal of Computer and System Sciences **55** (1997) 199–209
23. Hesse, W.: Dynamic Computational Complexity. PhD thesis, Department of Computer Science, UMass, Amherst (2003)
24. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: European Symp. On Programming. (2003) 380–398
25. Hendren, L., Hummel, J., Nicolau, A.: Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In: SIGPLAN Conf. on Prog. Lang. Design and Impl., New York, NY, ACM Press (1992) 249–260